
What's New in Python

Release 3.10.2

A. M. Kuchling

January 18, 2022

Python Software Foundation
Email: docs@python.org

Contents

1	Summary – Release highlights	3
2	New Features	3
2.1	Parenthesized context managers	3
2.2	Better error messages	4
2.3	PEP 626: Precise line numbers for debugging and other tools	8
2.4	PEP 634: Structural Pattern Matching	8
2.5	Optional EncodingWarning and encoding="locale" option	12
3	New Features Related to Type Hints	12
3.1	PEP 604: New Type Union Operator	12
3.2	PEP 612: Parameter Specification Variables	13
3.3	PEP 613: TypeAlias	13
3.4	PEP 647: User-Defined Type Guards	13
4	Other Language Changes	14
5	New Modules	15
6	Improved Modules	15
6.1	asyncio	15
6.2	argparse	15
6.3	array	15
6.4	asynchat, asyncore, smtpd	15
6.5	base64	15
6.6	bdb	15
6.7	bisect	15
6.8	codecs	16
6.9	collections.abc	16
6.10	contextlib	16
6.11	curses	16
6.12	dataclasses	16
6.13	distutils	17
6.14	doctest	17
6.15	encodings	17
6.16	fileinput	18
6.17	faulthandler	18
6.18	gc	18

6.19 glob	18
6.20 hashlib	18
6.21 hmac	18
6.22 IDLE and idlelib	18
6.23 importlib.metadata	19
6.24 inspect	19
6.25 linecache	19
6.26 os	19
6.27 os.path	20
6.28 pathlib	20
6.29 platform	20
6.30 pprint	20
6.31 py_compile	20
6.32 pyclbr	20
6.33 shelve	20
6.34 statistics	21
6.35 site	21
6.36 socket	21
6.37 ssl	21
6.38 sqlite3	22
6.39 sys	22
6.40 _thread	22
6.41 threading	22
6.42 traceback	22
6.43 types	22
6.44 typing	22
6.45 unittest	23
6.46 urllib.parse	23
6.47 xml	23
6.48 zipimport	24
7 Optimizations	24
8 Deprecated	25
9 Removed	27
10 Porting to Python 3.10	28
10.1 Changes in the Python syntax	28
10.2 Changes in the Python API	28
10.3 Changes in the C API	29
11 CPython bytecode changes	29
12 Build Changes	29
13 C API Changes	30
13.1 PEP 652: Maintaining the Stable ABI	30
13.2 New Features	30
13.3 Porting to Python 3.10	31
13.4 Deprecated	32
13.5 Removed	32
Index	35

Release 3.10.2

Date January 18, 2022

Editor Pablo Galindo Salgado

This article explains the new features in Python 3.10, compared to 3.9.

For full details, see the [changelog](#).

1 Summary – Release highlights

New syntax features:

- [PEP 634](#), Structural Pattern Matching: Specification
- [PEP 635](#), Structural Pattern Matching: Motivation and Rationale
- [PEP 636](#), Structural Pattern Matching: Tutorial
- [bpo-12782](#), Parenthesized context managers are now officially allowed.

New features in the standard library:

- [PEP 618](#), Add Optional Length-Checking To zip.

Interpreter improvements:

- [PEP 626](#), Precise line numbers for debugging and other tools.

New typing features:

- [PEP 604](#), Allow writing union types as X | Y
- [PEP 613](#), Explicit Type Aliases
- [PEP 612](#), Parameter Specification Variables

Important deprecations, removals or restrictions:

- [PEP 644](#), Require OpenSSL 1.1.1 or newer
- [PEP 632](#), Deprecate distutils module.
- [PEP 623](#), Deprecate and prepare for the removal of the wstr member in PyUnicodeObject.
- [PEP 624](#), Remove Py_UNICODE encoder APIs
- [PEP 597](#), Add optional EncodingWarning

2 New Features

2.1 Parenthesized context managers

Using enclosing parentheses for continuation across multiple lines in context managers is now supported. This allows formatting a long collection of context managers in multiple lines in a similar way as it was previously possible with import statements. For instance, all these examples are now valid:

```
with (CtxManager() as example):
    ...
with (
    CtxManager1(),
    CtxManager2()
):
    ...
with (CtxManager1() as example,
```

(continues on next page)

(continued from previous page)

```
CtxManager2() :  
...  
  
with (CtxManager1(),  
      CtxManager2() as example):  
...  
  
with (  
      CtxManager1() as example1,  
      CtxManager2() as example2  
):  
...  
...
```

it is also possible to use a trailing comma at the end of the enclosed group:

```
with (  
      CtxManager1() as example1,  
      CtxManager2() as example2,  
      CtxManager3() as example3,  
):  
...  
...
```

This new syntax uses the non LL(1) capacities of the new parser. Check [PEP 617](#) for more details.

(Contributed by Guido van Rossum, Pablo Galindo and Lysandros Nikolaou in [bpo-12782](#) and [bpo-40334](#).)

2.2 Better error messages

SyntaxErrors

When parsing code that contains unclosed parentheses or brackets the interpreter now includes the location of the unclosed bracket of parentheses instead of displaying *SyntaxError: unexpected EOF while parsing* or pointing to some incorrect location. For instance, consider the following code (notice the unclosed '{'):

```
expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4, 37: 4,  
            38: 4, 39: 4, 45: 5, 46: 5, 47: 5, 48: 5, 49: 5, 54: 6,  
some_other_code = foo()
```

Previous versions of the interpreter reported confusing places as the location of the syntax error:

```
File "example.py", line 3  
    some_other_code = foo()  
          ^  
  
SyntaxError: invalid syntax
```

but in Python 3.10 a more informative error is emitted:

```
File "example.py", line 1  
expected = {9: 1, 18: 2, 19: 2, 27: 3, 28: 3, 29: 3, 36: 4, 37: 4,  
           ^  
  
SyntaxError: '{' was never closed
```

In a similar way, errors involving unclosed string literals (single and triple quoted) now point to the start of the string instead of reporting EOF/EOL.

These improvements are inspired by previous work in the PyPy interpreter.

(Contributed by Pablo Galindo in [bpo-42864](#) and Batuhan Taskaya in [bpo-40176](#).)

`SyntaxError` exceptions raised by the interpreter will now highlight the full error range of the expression that constitutes the syntax error itself, instead of just where the problem is detected. In this way, instead of displaying (before Python 3.10):

```
>>> foo(x, z for z in range(10), t, w)
File "<stdin>", line 1
    foo(x, z for z in range(10), t, w)
          ^
SyntaxError: Generator expression must be parenthesized
```

now Python 3.10 will display the exception as:

```
>>> foo(x, z for z in range(10), t, w)
File "<stdin>", line 1
    foo(x, z for z in range(10), t, w)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
SyntaxError: Generator expression must be parenthesized
```

This improvement was contributed by Pablo Galindo in bpo-43914.

A considerable amount of new specialized messages for `SyntaxError` exceptions have been incorporated. Some of the most notable ones are as follows:

- Missing `:` before blocks:

```
>>> if rocket.position > event_horizon
File "<stdin>", line 1
    if rocket.position > event_horizon
          ^
SyntaxError: expected ':'
```

(Contributed by Pablo Galindo in bpo-42997)

- Unparenthesised tuples in comprehensions targets:

```
>>> {x,y for x,y in zip('abcd', '1234')}
File "<stdin>", line 1
    {x,y for x,y in zip('abcd', '1234')}
          ^
SyntaxError: did you forget parentheses around the comprehension_
→target?
```

(Contributed by Pablo Galindo in bpo-43017)

- Missing commas in collection literals and between expressions:

```
>>> items = {
...     x: 1,
...     y: 2
...     z: 3,
File "<stdin>", line 3
    y: 2
          ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

(Contributed by Pablo Galindo in bpo-43822)

- Multiple Exception types without parentheses:

```
>>> try:
...     build_dyson_sphere()
... except NotEnoughScienceError, NotEnoughResourcesError:
File "<stdin>", line 3
    except NotEnoughScienceError, NotEnoughResourcesError:
```

(continues on next page)

```
^
SyntaxError: multiple exception types must be parenthesized
```

(Contributed by Pablo Galindo in bpo-43149)

- Missing : and values in dictionary literals:

```
>>> values = {
...     x: 1,
...     y: 2,
...     z:
... }
File "<stdin>", line 4
    z:
    ^
SyntaxError: expression expected after dictionary key and ':'

>>> values = {x:1, y:2, z w:3}
File "<stdin>", line 1
    values = {x:1, y:2, z w:3}
                           ^
SyntaxError: ':' expected after dictionary key
```

(Contributed by Pablo Galindo in bpo-43823)

- try blocks without except or finally blocks:

```
>>> try:
...     x = 2
... something = 3
File "<stdin>", line 3
    something = 3
    ^^^^^^^^^^
SyntaxError: expected 'except' or 'finally' block
```

(Contributed by Pablo Galindo in bpo-44305)

- Usage of = instead of == in comparisons:

```
>>> if rocket.position = event_horizon:
File "<stdin>", line 1
    if rocket.position = event_horizon:
        ^
SyntaxError: cannot assign to attribute here. Maybe you meant '=='?
→ instead of '='?
```

(Contributed by Pablo Galindo in bpo-43797)

- Usage of * in f-strings:

```
>>> f"Black holes {*all_black_holes} and revelations"
File "<stdin>", line 1
    (*all_black_holes)
    ^
SyntaxError: f-string: cannot use starred expression here
```

(Contributed by Pablo Galindo in bpo-41064)

IndentationErrors

Many `IndentationError` exceptions now have more context regarding what kind of block was expecting an indentation, including the location of the statement:

```
>>> def foo():
...     if lel:
...         x = 2
File "<stdin>", line 3
      x = 2
^
IndentationError: expected an indented block after 'if' statement in line 2
```

AttributeErrors

When printing `AttributeError`, `PyErr_Display()` will offer suggestions of similar attribute names in the object that the exception was raised from:

```
>>> collections.namedtoplo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'collections' has no attribute 'namedtoplo'. Did you mean:_
 →namedtuple?
```

(Contributed by Pablo Galindo in [bpo-38530](#).)

Warning: Notice this won't work if `PyErr_Display()` is not called to display the error which can happen if some other custom error display function is used. This is a common scenario in some REPLs like IPython.

NameErrors

When printing `NameError` raised by the interpreter, `PyErr_Display()` will offer suggestions of similar variable names in the function that the exception was raised from:

```
>>> schwarzschild_black_hole = None
>>> schwarschild_black_hole
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'schwarschild_black_hole' is not defined. Did you mean:_
 →schwarzschild_black_hole?
```

(Contributed by Pablo Galindo in [bpo-38530](#).)

Warning: Notice this won't work if `PyErr_Display()` is not called to display the error, which can happen if some other custom error display function is used. This is a common scenario in some REPLs like IPython.

2.3 PEP 626: Precise line numbers for debugging and other tools

PEP 626 brings more precise and reliable line numbers for debugging, profiling and coverage tools. Tracing events, with the correct line number, are generated for all lines of code executed and only for lines of code that are executed.

The `f_lineno` attribute of frame objects will always contain the expected line number.

The `co_lnotab` attribute of code objects is deprecated and will be removed in 3.12. Code that needs to convert from offset to line number should use the new `co_lines()` method instead.

2.4 PEP 634: Structural Pattern Matching

Structural pattern matching has been added in the form of a *match statement* and *case statements* of patterns with associated actions. Patterns consist of sequences, mappings, primitive data types as well as class instances. Pattern matching enables programs to extract information from complex data types, branch on the structure of data, and apply specific actions based on different forms of data.

Syntax and operations

The generic syntax of pattern matching is:

```
match subject:  
    case <pattern_1>:  
        <action_1>  
    case <pattern_2>:  
        <action_2>  
    case <pattern_3>:  
        <action_3>  
    case _:  
        <action_wildcard>
```

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. Specifically, pattern matching operates by:

1. using data with type and shape (the `subject`)
2. evaluating the `subject` in the `match` statement
3. comparing the `subject` with each pattern in a `case` statement from top to bottom until a match is confirmed.
4. executing the action associated with the pattern of the confirmed match
5. If an exact match is not confirmed, the last case, a wildcard `_`, if provided, will be used as the matching case.
If an exact match is not confirmed and a wildcard case does not exist, the entire match block is a no-op.

Declarative approach

Readers may be aware of pattern matching through the simple example of matching a subject (data object) to a literal (pattern) with the switch statement found in C, Java or JavaScript (and many other languages). Often the switch statement is used for comparison of an object/expression with case statements containing literals.

More powerful examples of pattern matching can be found in languages such as Scala and Elixir. With structural pattern matching, the approach is “declarative” and explicitly states the conditions (the patterns) for data to match.

While an “imperative” series of instructions using nested “if” statements could be used to accomplish something similar to structural pattern matching, it is less clear than the “declarative” approach. Instead the “declarative” approach states the conditions to meet for a match and is more readable through its explicit patterns. While structural pattern matching can be used in its simplest form comparing a variable to a literal in a case statement, its true value for Python lies in its handling of the subject’s type and shape.

Simple pattern: match to a literal

Let's look at this example as pattern matching in its simplest form: a value, the subject, being matched to several literals, the patterns. In the example below, `status` is the subject of the match statement. The patterns are each of the case statements, where literals represent request status codes. The associated action to the case is executed after a match:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

If the above function is passed a `status` of 418, “I’m a teapot” is returned. If the above function is passed a `status` of 500, the case statement with `_` will match as a wildcard, and “Something’s wrong with the internet” is returned. Note the last block: the variable name, `_`, acts as a *wildcard* and insures the subject will always match. The use of `_` is optional.

You can combine several literals in a single pattern using `|` (“or”):

```
case 401 | 403 | 404:
    return "Not allowed"
```

Behavior without the wildcard

If we modify the above example by removing the last case block, the example becomes:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
```

Without the use of `_` in a case statement, a match may not exist. If no match exists, the behavior is a no-op. For example, if `status` of 500 is passed, a no-op occurs.

Patterns with a literal and variable

Patterns can look like unpacking assignments, and a pattern may be used to bind variables. In this example, a data point can be unpacked to its x-coordinate and y-coordinate:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
```

(continues on next page)

```
case _:
    raise ValueError("Not a point")
```

The first pattern has two literals, `(0, 0)`, and may be thought of as an extension of the literal pattern shown above. The next two patterns combine a literal and a variable, and the variable *binds* a value from the subject (`point`). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment `(x, y) = point`.

Patterns and classes

If you are using classes to structure your data, you can use as a pattern the class name followed by an argument list resembling a constructor. This pattern has the ability to capture class attributes into variables:

```
class Point:
    x: int
    y: int

def location(point):
    match point:
        case Point(x=0, y=0):
            print("Origin is the point's location.")
        case Point(x=0, y=y):
            print(f"Y={y} and the point is on the y-axis.")
        case Point(x=x, y=0):
            print(f"X={x} and the point is on the x-axis.")
        case Point():
            print("The point is located somewhere else on the plane.")
        case _:
            print("Not a point")
```

Patterns with positional parameters

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. data-classes). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to `("x", "y")`, the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Nested patterns

Patterns can be arbitrarily nested. For example, if our data is a short list of points, it could be matched like this:

```
match points:
    case []:
        print("No points in the list.")
    case [Point(0, 0)]:
        print("The origin is the only point in the list.")
    case [Point(x, y)]:
        print(f"A single point {x}, {y} is in the list.")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two points on the Y axis at {y1}, {y2} are in the list.")
```

(continues on next page)

```
case _:
    print("Something else is found in the list.")
```

Complex patterns and the wildcard

To this point, the examples have used `_` alone in the last case statement. A wildcard can be used in more complex patterns, such as `('error', code, _)`. For example:

```
match test_variable:
    case ('warning', code, 40):
        print("A warning has been received.")
    case ('error', code, _):
        print(f"An error {code} occurred.")
```

In the above case, `test_variable` will match for `('error', code, 100)` and `('error', code, 800)`.

Guard

We can add an `if` clause to a pattern, known as a “guard”. If the guard is false, `match` goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at {x}.")
    case Point(x, y):
        print(f"Point is not on the diagonal.")
```

Other Key Features

Several other key features:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. Technically, the subject must be a sequence. Therefore, an important exception is that patterns don't match iterators. Also, to prevent a common mistake, sequence patterns don't match strings.
- Sequence patterns support wildcards: `[x, y, *rest]` and `(x, y, *rest)` work similar to wildcards in unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.
- Mapping patterns: `{ "bandwidth": b, "latency": l }` captures the `"bandwidth"` and `"latency"` values from a dict. Unlike sequence patterns, extra keys are ignored. A wildcard `**rest` is also supported. (But `**_` would be redundant, so is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

This binds `x1`, `y1`, `x2`, `y2` like you would expect without the `as` clause, and `p2` to the entire second item of the subject.

- Most literals are compared by equality. However, the singletons `True`, `False` and `None` are compared by identity.
- Named constants may be used in patterns. These named constants must be dotted names to prevent the constant from being interpreted as a capture variable:

```

from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")

```

For the full specification see [PEP 634](#). Motivation and rationale are in [PEP 635](#), and a longer tutorial is in [PEP 636](#).

2.5 Optional EncodingWarning and encoding="locale" option

The default encoding of `TextIOWrapper` and `open()` is platform and locale dependent. Since UTF-8 is used on most Unix platforms, omitting `encoding` option when opening UTF-8 files (e.g. JSON, YAML, TOML, Markdown) is a very common bug. For example:

```

# BUG: "rb" mode or encoding="utf-8" should be used.
with open("data.json") as f:
    data = json.load(f)

```

To find this type of bug, an optional `EncodingWarning` is added. It is emitted when `sys.flags.warn_default_encoding` is true and locale-specific default encoding is used.

`-X warn_default_encoding` option and `PYTHONWARNDEFAULTENCODING` are added to enable the warning.

See `io-text-encoding` for more information.

3 New Features Related to Type Hints

This section covers major changes affecting [PEP 484](#) type hints and the `typing` module.

3.1 PEP 604: New Type Union Operator

A new type union operator was introduced which enables the syntax `X | Y`. This provides a cleaner way of expressing ‘either type X or type Y’ instead of using `typing.Union`, especially in type hints.

In previous versions of Python, to apply a type hint for functions accepting arguments of multiple types, `typing.Union` was used:

```

def square(number: Union[int, float]) -> Union[int, float]:
    return number ** 2

```

Type hints can now be written in a more succinct manner:

```

def square(number: int | float) -> int | float:
    return number ** 2

```

This new syntax is also accepted as the second argument to `isinstance()` and `issubclass()`:

```

>>> isinstance(1, int | str)
True

```

See types-union and [PEP 604](#) for more details.

(Contributed by Maggie Moss and Philippe Prados in [bpo-41428](#), with additions by Yurii Karabas and Serhiy Storchaka in [bpo-44490](#).)

3.2 PEP 612: Parameter Specification Variables

Two new options to improve the information provided to static type checkers for [PEP 484](#)'s `Callable` have been added to the `typing` module.

The first is the parameter specification variable. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. Examples of usage can be found in `typing.ParamSpec`. Previously, there was no easy way to type annotate dependency of parameter types in such a precise manner.

The second option is the new `Concatenate` operator. It's used in conjunction with parameter specification variables to type annotate a higher order callable which adds or removes parameters of another callable. Examples of usage can be found in `typing.Concatenate`.

See `typing.Callable`, `typing.ParamSpec`, `typing.Concatenate`, `typing.ParamSpecArgs`, `typing.ParamSpecKwargs`, and [PEP 612](#) for more details.

(Contributed by Ken Jin in [bpo-41559](#), with minor enhancements by Jelle Zijlstra in [bpo-43783](#). PEP written by Mark Mendoza.)

3.3 PEP 613: TypeAlias

[PEP 484](#) introduced the concept of type aliases, only requiring them to be top-level unannotated assignments. This simplicity sometimes made it difficult for type checkers to distinguish between type aliases and ordinary assignments, especially when forward references or invalid types were involved. Compare:

```
StrCache = 'Cache[str]' # a type alias
LOG_PREFIX = 'LOG[DEBUG]' # a module constant
```

Now the `typing` module has a special value `TypeAlias` which lets you declare type aliases more explicitly:

```
StrCache: TypeAlias = 'Cache[str]' # a type alias
LOG_PREFIX = 'LOG[DEBUG]' # a module constant
```

See [PEP 613](#) for more details.

(Contributed by Mikhail Golubev in [bpo-41923](#).)

3.4 PEP 647: User-Defined Type Guards

`TypeGuard` has been added to the `typing` module to annotate type guard functions and improve information provided to static type checkers during type narrowing. For more information, please see `TypeGuard`'s documentation, and [PEP 647](#).

(Contributed by Ken Jin and Guido van Rossum in [bpo-43766](#). PEP written by Eric Traut.)

4 Other Language Changes

- The `int` type has a new method `int.bit_count()`, returning the number of ones in the binary expansion of a given integer, also known as the population count. (Contributed by Niklas Fiekas in [bpo-29882](#).)
- The views returned by `dict.keys()`, `dict.values()` and `dict.items()` now all have a `mapping` attribute that gives a `types.MappingProxyType` object wrapping the original dictionary. (Contributed by Dennis Sweeney in [bpo-40890](#).)
- **PEP 618:** The `zip()` function now has an optional `strict` flag, used to require that all the iterables have an equal length.
- Builtin and extension functions that take integer arguments no longer accept `Decimals`, `Fractions` and other objects that can be converted to integers only with a loss (e.g. that have the `__int__()` method but do not have the `__index__()` method). (Contributed by Serhiy Storchaka in [bpo-37999](#).)
- If `object.__ipow__()` returns `NotImplemented`, the operator will correctly fall back to `object.__pow__()` and `object.__rpow__()` as expected. (Contributed by Alex Shkop in [bpo-38302](#).)
- Assignment expressions can now be used unparenthesized within set literals and set comprehensions, as well as in sequence indexes (but not slices).
- Functions have a new `__builtins__` attribute which is used to look for builtin symbols when a function is executed, instead of looking into `__globals__['__builtins__']`. The attribute is initialized from `__globals__['__builtins__']` if it exists, else from the current builtins. (Contributed by Mark Shannon in [bpo-42990](#).)
- Two new builtin functions – `aiter()` and `anext()` have been added to provide asynchronous counterparts to `iter()` and `next()`, respectively. (Contributed by Joshua Bronson, Daniel Pope, and Justin Wang in [bpo-31861](#).)
- Static methods (`@staticmethod`) and class methods (`@classmethod`) now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__`, `__annotations__`) and have a new `__wrapped__` attribute. Moreover, static methods are now callable as regular functions. (Contributed by Victor Stinner in [bpo-43682](#).)
- Annotations for complex targets (everything beside simple name targets defined by **PEP 526**) no longer cause any runtime effects with `from __future__ import annotations`. (Contributed by Batuhan Taskaya in [bpo-42737](#).)
- Class and module objects now lazy-create empty annotations dicts on demand. The annotations dicts are stored in the object's `__dict__` for backwards compatibility. This improves the best practices for working with `__annotations__`; for more information, please see `annotations-howto`. (Contributed by Larry Hastings in [bpo-43901](#).)
- Annotations consist of `yield`, `yield from`, `await` or named expressions are now forbidden under `from __future__ import annotations` due to their side effects. (Contributed by Batuhan Taskaya in [bpo-42725](#).)
- Usage of unbound variables, `super()` and other expressions that might alter the processing of symbol table as annotations are now rendered effectless under `from __future__ import annotations`. (Contributed by Batuhan Taskaya in [bpo-42725](#).)
- Hashes of `NaN` values of both `float` type and `decimal.Decimal` type now depend on object identity. Formerly, they always hashed to 0 even though `NaN` values are not equal to one another. This caused potentially quadratic runtime behavior due to excessive hash collisions when creating dictionaries and sets containing multiple `NaN`s. (Contributed by Raymond Hettinger in [bpo-43475](#).)
- A `SyntaxError` (instead of a `NameError`) will be raised when deleting the `__debug__` constant. (Contributed by Dong-hee Na in [bpo-45000](#).)
- `SyntaxError` exceptions now have `end_lineno` and `end_offset` attributes. They will be `None` if not determined. (Contributed by Pablo Galindo in [bpo-43914](#).)

5 New Modules

- None yet.

6 Improved Modules

6.1 `asyncio`

Add missing `connect_accepted_socket()` method. (Contributed by Alex Grönholm in [bpo-41332](#).)

6.2 `argparse`

Misleading phrase “optional arguments” was replaced with “options” in argparse help. Some tests might require adaptation if they rely on exact output match. (Contributed by Raymond Hettinger in [bpo-9694](#).)

6.3 `array`

The `index()` method of `array.array` now has optional `start` and `stop` parameters. (Contributed by Anders Lorentsen and Zackery Spytz in [bpo-31956](#).)

6.4 `asynchat`, `asyncore`, `smtplib`

These modules have been marked as deprecated in their module documentation since Python 3.6. An import-time `DeprecationWarning` has now been added to all three of these modules.

6.5 `base64`

Add `base64.b32hexencode()` and `base64.b32hexdecode()` to support the Base32 Encoding with Extended Hex Alphabet.

6.6 `bdb`

Add `clearBreakpoints()` to reset all set breakpoints. (Contributed by Irit Katriel in [bpo-24160](#).)

6.7 `bisect`

Added the possibility of providing a `key` function to the APIs in the `bisect` module. (Contributed by Raymond Hettinger in [bpo-4356](#).)

6.8 codecs

Add a `codecs.unregister()` function to unregister a codec search function. (Contributed by Hai Shi in [bpo-41842](#).)

6.9 collections.abc

The `__args__` of the parameterized generic for `collections.abc.Callable` are now consistent with `typing.Callable`. `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. To allow this change, `types.GenericAlias` can now be subclassed, and a subclass will be returned when subscripting the `collections.abc.Callable` type. Note that a `TypeError` may be raised for invalid forms of parameterizing `collections.abc.Callable` which may have passed silently in Python 3.9. (Contributed by Ken Jin in [bpo-42195](#).)

6.10 contextlib

Add a `contextlib.aclosing()` context manager to safely close async generators and objects representing asynchronously released resources. (Contributed by Joongi Kim and John Belmonte in [bpo-41229](#).)

Add asynchronous context manager support to `contextlib.nullcontext()`. (Contributed by Tom Gringauz in [bpo-41543](#).)

Add `AsyncContextDecorator`, for supporting usage of async context managers as decorators.

6.11 curses

The extended color functions added in ncurses 6.1 will be used transparently by `curses.color_content()`, `curses.init_color()`, `curses.init_pair()`, and `curses.pair_content()`. A new function, `curses.has_extended_color_support()`, indicates whether extended color support is provided by the underlying ncurses library. (Contributed by Jeffrey Kintscher and Hans Petter Jansson in [bpo-36982](#).)

The `BUTTON5_*` constants are now exposed in the `curses` module if they are provided by the underlying curses library. (Contributed by Zackery Spytz in [bpo-39273](#).)

6.12 dataclasses

slots

Added `slots` parameter in `dataclasses.dataclass()` decorator. (Contributed by Yurii Karabas in [bpo-42269](#))

Keyword-only fields

`dataclasses` now supports fields that are keyword-only in the generated `__init__` method. There are a number of ways of specifying keyword-only fields.

You can say that every field is keyword-only:

```
from dataclasses import dataclass

@dataclass(kw_only=True)
class Birthday:
    name: str
    birthday: datetime.date
```

Both `name` and `birthday` are keyword-only parameters to the generated `__init__` method.

You can specify keyword-only on a per-field basis:

```
from dataclasses import dataclass

@dataclass
class Birthday:
    name: str
    birthday: datetime.date = field(kw_only=True)
```

Here only `birthday` is keyword-only. If you set `kw_only` on individual fields, be aware that there are rules about re-ordering fields due to keyword-only fields needing to follow non-keyword-only fields. See the full `dataclasses` documentation for details.

You can also specify that all fields following a `KW_ONLY` marker are keyword-only. This will probably be the most common usage:

```
from dataclasses import dataclass, KW_ONLY

@dataclass
class Point:
    x: float
    y: float
    _: KW_ONLY
    z: float = 0.0
    t: float = 0.0
```

Here, `z` and `t` are keyword-only parameters, while `x` and `y` are not. (Contributed by Eric V. Smith in [bpo-43532](#))

6.13 distutils

The entire `distutils` package is deprecated, to be removed in Python 3.12. Its functionality for specifying package builds has already been completely replaced by third-party packages `setuptools` and `packaging`, and most other commonly used APIs are available elsewhere in the standard library (such as `platform`, `shutil`, `subprocess` or `sysconfig`). There are no plans to migrate any other functionality from `distutils`, and applications that are using other functions should plan to make private copies of the code. Refer to [PEP 632](#) for discussion.

The `bdist_wininst` command deprecated in Python 3.8 has been removed. The `bdist_wheel` command is now recommended to distribute binary packages on Windows. (Contributed by Victor Stinner in [bpo-42802](#).)

6.14 doctest

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

6.15 encodings

`encodings.normalize_encoding()` now ignores non-ASCII characters. (Contributed by Hai Shi in [bpo-39337](#).)

6.16 `fileinput`

Add `encoding` and `errors` parameters in `fileinput.input()` and `fileinput.FileInput`. (Contributed by Inada Naoki in [bpo-43712](#).)

`fileinput.hook_compressed()` now returns `TextIOWrapper` object when `mode` is “r” and file is compressed, like uncompressed files. (Contributed by Inada Naoki in [bpo-5758](#).)

6.17 `faulthandler`

The `faulthandler` module now detects if a fatal error occurs during a garbage collector collection. (Contributed by Victor Stinner in [bpo-44466](#).)

6.18 `gc`

Add audit hooks for `gc.get_objects()`, `gc.get_referrers()` and `gc.get_referents()`. (Contributed by Pablo Galindo in [bpo-43439](#).)

6.19 `glob`

Add the `root_dir` and `dir_fd` parameters in `glob()` and `iglob()` which allow to specify the root directory for searching. (Contributed by Serhiy Storchaka in [bpo-38144](#).)

6.20 `hashlib`

The `hashlib` module requires OpenSSL 1.1.1 or newer. (Contributed by Christian Heimes in [PEP 644](#) and [bpo-43669](#).)

The `hashlib` module has preliminary support for OpenSSL 3.0.0. (Contributed by Christian Heimes in [bpo-38820](#) and other issues.)

The pure-Python fallback of `pbkdf2_hmac()` is deprecated. In the future PBKDF2-HMAC will only be available when Python has been built with OpenSSL support. (Contributed by Christian Heimes in [bpo-43880](#).)

6.21 `hmac`

The `hmac` module now uses OpenSSL’s HMAC implementation internally. (Contributed by Christian Heimes in [bpo-40645](#).)

6.22 IDLE and `idlelib`

Make IDLE invoke `sys.excepthook()` (when started without ‘-n’). User hooks were previously ignored. (Patch by Ken Hilton in [bpo-43008](#).)

This change was backported to a 3.9 maintenance release.

Add a Shell sidebar. Move the primary prompt (‘>>>’) to the sidebar. Add secondary prompts (‘...’) to the sidebar. Left click and optional drag selects one or more lines of text, as with the editor line number sidebar. Right click after selecting text lines displays a context menu with ‘copy with prompts’. This zips together prompts from the sidebar with lines from the selected text. This option also appears on the context menu for the text. (Contributed by Tal Einat in [bpo-37903](#).)

Use spaces instead of tabs to indent interactive code. This makes interactive code entries ‘look right’. Making this feasible was a major motivation for adding the shell sidebar. Contributed by Terry Jan Reedy in [bpo-37892](#).)

We expect to backport these shell changes to a future 3.9 maintenance release.

Highlight the new soft keywords `match`, `case`, and `_` in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some `_`-s in `case` patterns. (Contributed by Tal Einat in [bpo-44010](#).)

6.23 importlib.metadata

Feature parity with `importlib_metadata` 4.6 ([history](#)).

`importlib.metadata` entry points now provide a nicer experience for selecting entry points by group and name through a new `importlib.metadata.EntryPoints` class. See the Compatibility Note in the docs for more info on the deprecation and usage.

Added `importlib.metadata.packages_distributions()` for resolving top-level Python modules and packages to their `importlib.metadata.Distribution`.

6.24 inspect

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

Add `inspect.get_annotations()`, which safely computes the annotations defined on an object. It works around the quirks of accessing the annotations on various types of objects, and makes very few assumptions about the object it examines. `inspect.get_annotations()` can also correctly un-stringize stringized annotations. `inspect.get_annotations()` is now considered best practice for accessing the annotations dict defined on any Python object; for more information on best practices for working with annotations, please see [annotations-howto](#). Relatedly, `inspect.signature()`, `inspect.Signature.from_callable()`, and `inspect.Signature.from_function()` now call `inspect.get_annotations()` to retrieve annotations. This means `inspect.signature()` and `inspect.Signature.from_callable()` can also now un-stringize stringized annotations. (Contributed by Larry Hastings in [bpo-43817](#).)

6.25 linecache

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

6.26 os

Add `os.cpu_count()` support for VxWorks RTOS. (Contributed by Peixing Xin in [bpo-41440](#).)

Add a new function `os.eventfd()` and related helpers to wrap the `eventfd2` syscall on Linux. (Contributed by Christian Heimes in [bpo-41001](#).)

Add `os.splice()` that allows to move data between two file descriptors without copying between kernel address space and user address space, where one of the file descriptors must refer to a pipe. (Contributed by Pablo Galindo in [bpo-41625](#).)

Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` for macOS. (Contributed by Dong-hee Na in [bpo-43106](#).)

6.27 os.path

`os.path.realpath()` now accepts a *strict* keyword-only argument. When set to True, `OSError` is raised if a path doesn't exist or a symlink loop is encountered. (Contributed by Barney Gale in [bpo-43757](#).)

6.28 pathlib

Add slice support to `PurePath.parents`. (Contributed by Joshua Cannon in [bpo-35498](#))

Add negative indexing support to `PurePath.parents`. (Contributed by Yaroslav Pankovych in [bpo-21041](#))

Add `Path.hardlink_to` method that supersedes `link_to()`. The new method has the same argument order as `symlink_to()`. (Contributed by Barney Gale in [bpo-39950](#).)

`pathlib.Path.stat()` and `chmod()` now accept a *follow_symlinks* keyword-only argument for consistency with corresponding functions in the `os` module. (Contributed by Barney Gale in [bpo-39906](#).)

6.29 platform

Add `platform.freedesktop_os_release()` to retrieve operation system identification from `freedesktop.org os-release` standard file. (Contributed by Christian Heimes in [bpo-28468](#))

6.30 pprint

`pprint.pprint()` now accepts a new `underscore_numbers` keyword argument. (Contributed by sblondon in [bpo-42914](#).)

`pprint` can now pretty-print `dataclasses.dataclass` instances. (Contributed by Lewis Gaul in [bpo-43080](#).)

6.31 py_compile

Add `--quiet` option to command-line interface of `py_compile`. (Contributed by Gregory Schevchenko in [bpo-38731](#).)

6.32 pycbr

Add an `end_lineno` attribute to the `Function` and `Class` objects in the tree returned by `pycbr.readline()` and `pycbr.readline_ex()`. It matches the existing (start) `lineno`. (Contributed by Aviral Srivastava in [bpo-38307](#).)

6.33 shelve

The `shelve` module now uses `pickle.DEFAULT_PROTOCOL` by default instead of `pickle` protocol 3 when creating shelves. (Contributed by Zackery Spytz in [bpo-34204](#).)

6.34 statistics

Add `covariance()`, Pearson's `correlation()`, and simple `linear_regression()` functions. (Contributed by Tymoteusz Wołodźko in [bpo-38490](#).)

6.35 site

When a module does not define `__loader__`, fall back to `__spec__.loader`. (Contributed by Brett Cannon in [bpo-42133](#).)

6.36 socket

The exception `socket.timeout` is now an alias of `TimeoutError`. (Contributed by Christian Heimes in [bpo-42413](#).)

Add option to create MPTCP sockets with `IPPROTO_MPTCP` (Contributed by Rui Cunha in [bpo-43571](#).)

Add `IP_RECVTOS` option to receive the type of service (ToS) or DSCP/ECN fields (Contributed by Georg Sauthoff in [bpo-44077](#).)

6.37 ssl

The `ssl` module requires OpenSSL 1.1.1 or newer. (Contributed by Christian Heimes in [PEP 644](#) and [bpo-43669](#).)

The `ssl` module has preliminary support for OpenSSL 3.0.0 and new option `OP_IGNORE_UNEXPECTED_EOF`. (Contributed by Christian Heimes in [bpo-38820](#), [bpo-43794](#), [bpo-43788](#), [bpo-43791](#), [bpo-43799](#), [bpo-43920](#), [bpo-43789](#), and [bpo-43811](#).)

Deprecated function and use of deprecated constants now result in a `DeprecationWarning`. `ssl.SSLContext.options` has `OP_NO_SSLv2` and `OP_NO_SSLv3` set by default and therefore cannot warn about setting the flag again. The [deprecation section](#) has a list of deprecated features. (Contributed by Christian Heimes in [bpo-43880](#).)

The `ssl` module now has more secure default settings. Ciphers without forward secrecy or SHA-1 MAC are disabled by default. Security level 2 prohibits weak RSA, DH, and ECC keys with less than 112 bits of security. `SSLContext` defaults to minimum protocol version TLS 1.2. Settings are based on Hynek Schlawack's research. (Contributed by Christian Heimes in [bpo-43998](#).)

The deprecated protocols SSL 3.0, TLS 1.0, and TLS 1.1 are no longer officially supported. Python does not block them actively. However OpenSSL build options, distro configurations, vendor patches, and cipher suites may prevent a successful handshake.

Add a `timeout` parameter to the `ssl.get_server_certificate()` function. (Contributed by Zackery Spytz in [bpo-31870](#).)

The `ssl` module uses heap-types and multi-phase initialization. (Contributed by Christian Heimes in [bpo-42333](#).)

A new verify flag `VERIFY_X509_PARTIAL_CHAIN` has been added. (Contributed by l0x in [bpo-40849](#).)

6.38 sqlite3

Add audit events for `connect/handle()`, `enable_load_extension()`, and `load_extension()`. (Contributed by Erlend E. Aasland in [bpo-43762](#).)

6.39 sys

Add `sys.orig_argv` attribute: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in [bpo-23427](#).)

Add `sys.stdlib_module_names`, containing the list of the standard library module names. (Contributed by Victor Stinner in [bpo-42955](#).)

6.40 _thread

`_thread.interrupt_main()` now takes an optional signal number to simulate (the default is still `signal.SIGINT`). (Contributed by Antoine Pitrou in [bpo-43356](#).)

6.41 threading

Add `threading.gettrace()` and `threading.getprofile()` to retrieve the functions set by `threading.settrace()` and `threading.setprofile()` respectively. (Contributed by Mario Corchero in [bpo-42251](#).)

Add `threading.__excepthook__` to allow retrieving the original value of `threading.excepthook()` in case it is set to a broken or a different value. (Contributed by Mario Corchero in [bpo-42308](#).)

6.42 traceback

The `format_exception()`, `format_exception_only()`, and `print_exception()` functions can now take an exception object as a positional-only argument. (Contributed by Zackery Spytz and Matthias Bussonnier in [bpo-26389](#).)

6.43 types

Reintroduce the `types.EllipsisType`, `types.NoneType` and `types.NotImplementedType` classes, providing a new set of types readily interpretable by type checkers. (Contributed by Bas van Beek in [bpo-41810](#).)

6.44 typing

For major changes, see [*New Features Related to Type Hints*](#).

The behavior of `typing.Literal` was changed to conform with [PEP 586](#) and to match the behavior of static type checkers specified in the PEP.

1. `Literal` now de-duplicates parameters.
2. Equality comparisons between `Literal` objects are now order independent.
3. `Literal` comparisons now respect types. For example, `Literal[0] == Literal[False]` previously evaluated to `True`. It is now `False`. To support this change, the internally used type cache now supports differentiating types.
4. `Literal` objects will now raise a `TypeError` exception during equality comparisons if any of their parameters are not hashable. Note that declaring `Literal` with unhashable parameters will not throw an error:

```
>>> from typing import Literal
>>> Literal[{}]
>>> Literal[{}] == Literal[{False}]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

(Contributed by Yurii Karabas in [bpo-42345](#).)

Add new function `typing.is_typeddict()` to introspect if an annotation is a `typing.TypedDict`. (Contributed by Patrick Reader in [bpo-41792](#))

Subclasses of `typing.Protocol` which only have data variables declared will now raise a `TypeError` when checked with `isinstance` unless they are decorated with `runtime_checkable()`. Previously, these checks passed silently. Users should decorate their subclasses with the `runtime_checkable()` decorator if they want runtime protocols. (Contributed by Yurii Karabas in [bpo-38908](#))

Importing from the `typing.io` and `typing.re` submodules will now emit `DeprecationWarning`. These submodules have been deprecated since Python 3.8 and will be removed in a future version of Python. Anything belonging to those submodules should be imported directly from `typing` instead. (Contributed by Sebastian Rittau in [bpo-38291](#))

6.45 unittest

Add new method `assertNoLogs()` to complement the existing `assertLogs()`. (Contributed by Kit Yan Choi in [bpo-39385](#).)

6.46 urllib.parse

Python versions earlier than Python 3.10 allowed using both ; and & as query parameter separators in `urllib.parse.parse_qs()` and `urllib.parse.parse_qsl()`. Due to security concerns, and to conform with newer W3C recommendations, this has been changed to allow only a single separator key, with & as the default. This change also affects `cgi.parse()` and `cgi.parse_multipart()` as they use the affected functions internally. For more details, please see their respective documentation. (Contributed by Adam Goldschmidt, Senthil Kumaran and Ken Jin in [bpo-42967](#).)

The presence of newline or tab characters in parts of a URL allows for some forms of attacks. Following the WHATWG specification that updates [RFC 3986](#), ASCII newline \n, \r and tab \t characters are stripped from the URL by the parser in `urllib.parse` preventing such attacks. The removal characters are controlled by a new module level variable `urllib.parse._UNSAFE_URL_BYT`ES_TO_REMOVE. (See [bpo-43882](#))

6.47 xml

Add a `LexicalHandler` class to the `xml.sax.handler` module. (Contributed by Jonathan Gossage and Zackery Spytz in [bpo-35018](#).)

6.48 zipimport

Add methods related to [PEP 451](#): `find_spec()`, `zipimport.zipimporter.create_module()`, and `zipimport.zipimporter.exec_module()`. (Contributed by Brett Cannon in [bpo-42131](#).)

Add `invalidate_caches()` method. (Contributed by Desmond Cheong in [bpo-14678](#).)

7 Optimizations

- Constructors `str()`, `bytes()` and `bytearray()` are now faster (around 30–40% for small objects). (Contributed by Serhiy Storchaka in [bpo-41334](#).)
- The `runpy` module now imports fewer modules. The `python3 -m module-name` command startup time is 1.4x faster in average. On Linux, `python3 -I -m module-name` imports 69 modules on Python 3.9, whereas it only imports 51 modules (-18) on Python 3.10. (Contributed by Victor Stinner in [bpo-41006](#) and [bpo-41718](#).)
- The `LOAD_ATTR` instruction now uses new “per opcode cache” mechanism. It is about 36% faster now for regular attributes and 44% faster for slots. (Contributed by Pablo Galindo and Yury Selivanov in [bpo-42093](#) and Guido van Rossum in [bpo-42927](#), based on ideas implemented originally in PyPy and MicroPython.)
- When building Python with `--enable-optimizations` now `-fno-semantic-interposition` is added to both the compile and link line. This speeds builds of the Python interpreter created with `--enable-shared` with `gcc` by up to 30%. See [this article](#) for more details. (Contributed by Victor Stinner and Pablo Galindo in [bpo-38980](#).)
- Use a new output buffer management code for `bz2/lzma/zlib` modules, and add `.readall()` function to `_compression.DecompressReader` class. `bz2` decompression is now 1.09x ~ 1.17x faster, `lzma` decompression 1.20x ~ 1.32x faster, `GzipFile.read(-1)` 1.11x ~ 1.18x faster. (Contributed by Ma Lin, reviewed by Gregory P. Smith, in [bpo-41486](#))
- When using stringized annotations, annotations dicts for functions are no longer created when the function is created. Instead, they are stored as a tuple of strings, and the function object lazily converts this into the annotations dict on demand. This optimization cuts the CPU time needed to define an annotated function by half. (Contributed by Yurii Karabas and Inada Naoki in [bpo-42202](#))
- Substring search functions such as `str1` in `str2` and `str2.find(str1)` now sometimes use Crochemore & Perrin’s “Two-Way” string searching algorithm to avoid quadratic behavior on long strings. (Contributed by Dennis Sweeney in [bpo-41972](#))
- Add micro-optimizations to `_PyType_Lookup()` to improve type attribute cache lookup performance in the common case of cache hits. This makes the interpreter 1.04 times faster on average. (Contributed by Dino Viehland in [bpo-43452](#))
- The following built-in functions now support the faster [PEP 590](#) vectorcall calling convention: `map()`, `filter()`, `reversed()`, `bool()` and `float()`. (Contributed by Dong-hee Na and Jeroen Demeyer in [bpo-43575](#), [bpo-43287](#), [bpo-41922](#), [bpo-41873](#) and [bpo-41870](#))
- `BZ2File` performance is improved by removing internal `RLOCK`. This makes `BZ2File` thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in `gzip` and `lzma` have always been. (Contributed by Inada Naoki in [bpo-43785](#)).

8 Deprecated

- Currently Python accepts numeric literals immediately followed by keywords, for example `0in x, 1or x, 0if 1else 2`. It allows confusing and ambiguous expressions like `[0x1for x in y]` (which can be interpreted as `[0x1 for x in y]` or `[0x1f or x in y]`). Starting in this release, a deprecation warning is raised if the numeric literal is immediately followed by one of keywords and, else, for, if, in, is and or. In future releases it will be changed to syntax warning, and finally to syntax error. (Contributed by Serhiy Storchaka in [bpo-43833](#).)
- Starting in this release, there will be a concerted effort to begin cleaning up old import semantics that were kept for Python 2.7 compatibility. Specifically, `find_loader()/find_module()` (superseded by `find_spec()`), `load_module()` (superseded by `exec_module()`), `module_repr()` (which the import system takes care of for you), the `__package__` attribute (superseded by `__spec__.parent`), the `__loader__` attribute (superseded by `__spec__.loader`), and the `__cached__` attribute (superseded by `__spec__.cached`) will slowly be removed (as well as other classes and methods in `importlib`). `ImportWarning` and/or `DeprecationWarning` will be raised as appropriate to help identify code which needs updating during this transition.
- The entire `distutils` namespace is deprecated, to be removed in Python 3.12. Refer to the *module changes* section for more information.
- Non-integer arguments to `random.randrange()` are deprecated. The `ValueError` is deprecated in favor of a `TypeError`. (Contributed by Serhiy Storchaka and Raymond Hettinger in [bpo-37319](#).)
- The various `load_module()` methods of `importlib` have been documented as deprecated since Python 3.6, but will now also trigger a `DeprecationWarning`. Use `exec_module()` instead. (Contributed by Brett Cannon in [bpo-26131](#).)
- `zimport.zipimporter.load_module()` has been deprecated in preference for `exec_module()`. (Contributed by Brett Cannon in [bpo-26131](#).)
- The use of `load_module()` by the import system now triggers an `ImportWarning` as `exec_module()` is preferred. (Contributed by Brett Cannon in [bpo-26131](#).)
- The use of `importlib.abc.MetaPathFinder.find_module()` and `importlib.abc.PathEntryFinder.find_module()` by the import system now trigger an `ImportWarning` as `importlib.abc.MetaPathFinder.find_spec()` and `importlib.abc.PathEntryFinder.find_spec()` are preferred, respectively. You can use `importlib.util.spec_from_loader()` to help in porting. (Contributed by Brett Cannon in [bpo-42134](#).)
- The use of `importlib.abc.PathEntryFinder.find_loader()` by the import system now triggers an `ImportWarning` as `importlib.abc.PathEntryFinder.find_spec()` is preferred. You can use `importlib.util.spec_from_loader()` to help in porting. (Contributed by Brett Cannon in [bpo-43672](#).)
- The various implementations of `importlib.abc.MetaPathFinder.find_module()` (`importlib.machinery.BuiltinImporter.find_module()`, `importlib.machinery.FrozenImporter.find_module()`, `importlib.machinery.WindowsRegistryFinder.find_module()`, `importlib.machinery.PathFinder.find_module()`, `importlib.abc.MetaPathFinder.find_module()`), `importlib.abc.PathEntryFinder.find_module()` (`importlib.machinery.FileFinder.find_module()`), and `importlib.abc.PathEntryFinder.find_loader()` (`importlib.machinery.FileFinder.find_loader()`) now raise `DeprecationWarning` and are slated for removal in Python 3.12 (previously they were documented as deprecated in Python 3.4). (Contributed by Brett Cannon in [bpo-42135](#).)
- `importlib.abc.Finder` is deprecated (including its sole method, `find_module()`). Both `importlib.abc.MetaPathFinder` and `importlib.abc.PathEntryFinder` no longer inherit from the class. Users should inherit from one of these two classes as appropriate instead. (Contributed by Brett Cannon in [bpo-42135](#).)
- The deprecations of `imp`, `importlib.find_loader()`, `importlib.util.set_package_wrapper()`, `importlib.util.set_loader_wrapper()`, `importlib.`

`util.module_for_loader()`, `pkgutil.ImpImporter`, and `pkgutil.ImpLoader` have all been updated to list Python 3.12 as the slated version of removal (they began raising `DeprecationWarning` in previous versions of Python). (Contributed by Brett Cannon in [bpo-43720](#).)

- The import system now uses the `__spec__` attribute on modules before falling back on `module_repr()` for a module's `__repr__()` method. Removal of the use of `module_repr()` is scheduled for Python 3.12. (Contributed by Brett Cannon in [bpo-42137](#).)
- `importlib.abc.Loader.module_repr()`, `importlib.machinery.FrozenLoader.module_repr()`, and `importlib.machinery.BuiltinLoader.module_repr()` are deprecated and slated for removal in Python 3.12. (Contributed by Brett Cannon in [bpo-42136](#).)
- `sqlite3.OptimizedUnicode` has been undocumented and obsolete since Python 3.3, when it was made an alias to `str`. It is now deprecated, scheduled for removal in Python 3.12. (Contributed by Erlend E. Aasland in [bpo-42264](#).)
- `asyncio.get_event_loop()` now emits a deprecation warning if there is no running event loop. In the future it will be an alias of `get_running_loop()`. `asyncio` functions which implicitly create `Future` or `Task` objects now emit a deprecation warning if there is no running event loop and no explicit `loop` argument is passed: `ensure_future()`, `wrap_future()`, `gather()`, `shield()`, `as_completed()` and constructors of `Future`, `Task`, `StreamReader`, `StreamReaderProtocol`. (Contributed by Serhiy Storchaka in [bpo-39529](#).)
- The undocumented built-in function `sqlite3.enable_shared_cache` is now deprecated, scheduled for removal in Python 3.12. Its use is strongly discouraged by the SQLite3 documentation. See [the SQLite3 docs](#) for more details. If a shared cache must be used, open the database in URI mode using the `cache=shared` query parameter. (Contributed by Erlend E. Aasland in [bpo-24464](#).)
- The following threading methods are now deprecated:
 - `threading.currentThread => threading.current_thread()`
 - `threading.activeCount => threading.active_count()`
 - `threading.Condition.notifyAll => threading.Condition.notify_all()`
 - `threading.Event.isSet => threading.Event.is_set()`
 - `threading.Thread.setName => threading.Thread.name`
 - `threading.thread.getName => threading.Thread.name`
 - `threading.Thread.isDaemon => threading.Thread.daemon`
 - `threading.Thread.setDaemon => threading.Thread.daemon`

(Contributed by Jelle Zijlstra in [bpo-21574](#).)

- `pathlib.Path.link_to()` is deprecated and slated for removal in Python 3.12. Use `pathlib.Path.hardlink_to()` instead. (Contributed by Barney Gale in [bpo-39950](#).)
- `cgi.log()` is deprecated and slated for removal in Python 3.12. (Contributed by Inada Naoki in [bpo-41139](#).)
- The following `ssl` features have been deprecated since Python 3.6, Python 3.7, or OpenSSL 1.1.0 and will be removed in 3.11:
 - `OP_NO_SSLv2`, `OP_NO_SSLv3`, `OP_NO_TLSv1`, `OP_NO_TLSv1_1`, `OP_NO_TLSv1_2`, and `OP_NO_TLSv1_3` are replaced by `sslSSLContext.minimum_version` and `sslSSLContext.maximum_version`.
 - `PROTOCOL_SSLv2`, `PROTOCOL_SSLv3`, `PROTOCOL_SSLv23`, `PROTOCOL_TLSv1`, `PROTOCOL_TLSv1_1`, `PROTOCOL_TLSv1_2`, and `PROTOCOL_TLS` are deprecated in favor of `PROTOCOL_TLS_CLIENT` and `PROTOCOL_TLS_SERVER`
 - `wrap_socket()` is replaced by `ssl.SSLContext.wrap_socket()`
 - `match_hostname()`
 - `RAND_pseudo_bytes()`, `RAND_egd()`

- NPN features like `ssl.SSLSocket.selected_npn_protocol()` and `ssl.SSLContext.set_npn_protocols()` are replaced by ALPN.
- The threading debug (`PYTHONTHREADDEBUG` environment variable) is deprecated in Python 3.10 and will be removed in Python 3.12. This feature requires a debug build of Python. (Contributed by Victor Stinner in [bpo-44584](#).)
- Importing from the `typing.io` and `typing.re` submodules will now emit `DeprecationWarning`. These submodules will be removed in a future version of Python. Anything belonging to these submodules should be imported directly from `typing` instead. (Contributed by Sebastian Rittau in [bpo-38291](#))

9 Removed

- Removed special methods `__int__`, `__float__`, `__floordiv__`, `__mod__`, `__divmod__`, `__rfloordiv__`, `__rmod__` and `__rdivmod__` of the `complex` class. They always raised a `TypeError`. (Contributed by Serhiy Storchaka in [bpo-41974](#).)
- The `ParserBase.error()` method from the private and undocumented `_markupbase` module has been removed. `html.parser.HTMLParser` is the only subclass of `ParserBase` and its `error()` implementation was already removed in Python 3.5. (Contributed by Berker Peksag in [bpo-31844](#).)
- Removed the `unicodedata.ucnhash_CAPI` attribute which was an internal PyCapsule object. The related private `_PyUnicode_Name_CAPI` structure was moved to the internal C API. (Contributed by Victor Stinner in [bpo-42157](#).)
- Removed the `parser` module, which was deprecated in 3.9 due to the switch to the new PEG parser, as well as all the C source and header files that were only being used by the old parser, including `node.h`, `parser.h`, `graminit.h` and `grammar.h`.
- Removed the Public C API functions `PyParser_SimpleParseStringFlags`, `PyParser_SimpleParseStringFlagsFilename`, `PyParser_SimpleParseFileFlags` and `PyNode_Compile` that were deprecated in 3.9 due to the switch to the new PEG parser.
- Removed the `formatter` module, which was deprecated in Python 3.4. It is somewhat obsolete, little used, and not tested. It was originally scheduled to be removed in Python 3.6, but such removals were delayed until after Python 2.7 EOL. Existing users should copy whatever classes they use into their code. (Contributed by Dong-hee Na and Terry J. Reedy in [bpo-42299](#).)
- Removed the `PyModule_GetWarningsModule()` function that was useless now due to the `_warnings` module was converted to a builtin module in 2.6. (Contributed by Hai Shi in [bpo-42599](#).)
- Remove deprecated aliases to collections-abstract-base-classes from the `collections` module. (Contributed by Victor Stinner in [bpo-37324](#).)
- The `loop` parameter has been removed from most of `asyncio`'s high-level API following deprecation in Python 3.8. The motivation behind this change is multifold:
 1. This simplifies the high-level API.
 2. The functions in the high-level API have been implicitly getting the current thread's running event loop since Python 3.7. There isn't a need to pass the event loop to the API in most normal use cases.
 3. Event loop passing is error-prone especially when dealing with loops running in different threads.

Note that the low-level API will still accept `loop`. See [*Changes in the Python API*](#) for examples of how to replace existing code.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in [bpo-42392](#).)

10 Porting to Python 3.10

This section lists previously described changes and other bugfixes that may require changes to your code.

10.1 Changes in the Python syntax

- Deprecation warning is now emitted when compiling previously valid syntax if the numeric literal is immediately followed by a keyword (like in `0 in x`). In future releases it will be changed to syntax warning, and finally to a syntax error. To get rid of the warning and make the code compatible with future releases just add a space between the numeric literal and the following keyword. (Contributed by Serhiy Storchaka in [bpo-43833](#).)

10.2 Changes in the Python API

- The `etype` parameters of the `format_exception()`, `format_exception_only()`, and `print_exception()` functions in the `traceback` module have been renamed to `exc`. (Contributed by Zackery Spytz and Matthias Bussonnier in [bpo-26389](#).)
- `atexit`: At Python exit, if a callback registered with `atexit.register()` fails, its exception is now logged. Previously, only some exceptions were logged, and the last exception was always silently ignored. (Contributed by Victor Stinner in [bpo-42639](#).)
- `collections.abc.Callable` generic now flattens type parameters, similar to what `typing.Callable` currently does. This means that `collections.abc.Callable[[int, str], str]` will have `__args__` of `(int, str, str)`; previously this was `([int, str], str)`. Code which accesses the arguments via `typing.get_args()` or `__args__` need to account for this change. Furthermore, `TypeError` may be raised for invalid forms of parameterizing `collections.abc.Callable` which may have passed silently in Python 3.9. (Contributed by Ken Jin in [bpo-42195](#).)
- `socket.htons()` and `socket.ntohs()` now raise `OverflowError` instead of `DeprecationWarning` if the given parameter will not fit in a 16-bit unsigned integer. (Contributed by Erlend E. Aasland in [bpo-42393](#).)
- The `loop` parameter has been removed from most of `asyncio`'s high-level API following deprecation in Python 3.8.

A coroutine that currently looks like this:

```
async def foo(loop):
    await asyncio.sleep(1, loop=loop)
```

Should be replaced with this:

```
async def foo():
    await asyncio.sleep(1)
```

If `foo()` was specifically designed *not* to run in the current thread's running event loop (e.g. running in another thread's event loop), consider using `asyncio.run_coroutine_threadsafe()` instead.

(Contributed by Yurii Karabas, Andrew Svetlov, Yury Selivanov and Kyle Stanley in [bpo-42392](#).)

- The `types.FunctionType` constructor now inherits the current builtins if the `globals` dictionary has no `"__builtins__"` key, rather than using `{"None": None}` as `builtins`: same behavior as `eval()` and `exec()` functions. Defining a function with `def function(...): ...` in Python is not affected, `globals` cannot be overridden with this syntax: it also inherits the current builtins. (Contributed by Victor Stinner in [bpo-42990](#).)

10.3 Changes in the C API

- The C API functions `PyParser_SimpleParseStringFlags`, `PyParser_SimpleParseStringFlagsFilename`, `PyParser_SimpleParseFileFlags`, `PyNode_Compile` and the type used by these functions, `struct _node`, were removed due to the switch to the new PEG parser.

Source should be now be compiled directly to a code object using, for example, `Py_CompileString()`. The resulting code object can then be evaluated using, for example, `PyEval_EvalCode()`.

Specifically:

- A call to `PyParser_SimpleParseStringFlags` followed by `PyNode_Compile` can be replaced by calling `Py_CompileString()`.
- There is no direct replacement for `PyParser_SimpleParseFileFlags`. To compile code from a `FILE *` argument, you will need to read the file in C and pass the resulting buffer to `Py_CompileString()`.
- To compile a file given a `char *` filename, explicitly open the file, read it and compile the result. One way to do this is using the `io` module with `PyImport_ImportModule()`, `PyObject_CallMethod()`, `PyBytes_AsString()` and `Py_CompileString()`, as sketched below. (Declarations and error handling are omitted.)

```
io_module = Import_ImportModule("io");
fileobject = PyObject_CallMethod(io_module, "open", "ss", filename, "rb");
source_bytes_object = PyObject_CallMethod(fileobject, "read", "");
result = PyObject_CallMethod(fileobject, "close", "");
source_buf = PyBytes_AsString(source_bytes_object);
code = Py_CompileString(source_buf, filename, Py_file_input);
```

- For `FrameObject` objects, the `f_lasti` member now represents a wordcode offset instead of a simple offset into the bytecode string. This means that this number needs to be multiplied by 2 to be used with APIs that expect a byte offset instead (like `PyCode_Addr2Line()` for example). Notice as well that the `f_lasti` member of `FrameObject` objects is not considered stable: please use `PyFrame_GetLineNumber()` instead.

11 CPython bytecode changes

- The `MAKE_FUNCTION` instruction now accepts either a dict or a tuple of strings as the function's annotations. (Contributed by Yurii Karabas and Inada Naoki in [bpo-42202](#))

12 Build Changes

- PEP 644**: Python now requires OpenSSL 1.1.1 or newer. OpenSSL 1.0.2 is no longer supported. (Contributed by Christian Heimes in [bpo-43669](#).)
- The C99 functions `sprintf()` and `vsnprintf()` are now required to build Python. (Contributed by Victor Stinner in [bpo-36020](#).)
- `sqlite3` requires SQLite 3.7.15 or higher. (Contributed by Sergey Fedoseev and Erlend E. Aasland in [bpo-40744](#) and [bpo-40810](#).)
- The `atexit` module must now always be built as a built-in module. (Contributed by Victor Stinner in [bpo-42639](#).)
- Add `--disable-test-modules` option to the `configure` script: don't build nor install test modules. (Contributed by Xavier de Gaye, Thomas Petazzoni and Peixing Xin in [bpo-27640](#).)

- Add `--with-wheel-pkg-dir=PATH` option to the `./configure` script. If specified, the `ensurepip` module looks for `setuptools` and `pip` wheel packages in this directory: if both are present, these wheel packages are used instead of `ensurepip` bundled wheel packages.

Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the `/usr/share/python-wheels/` directory and don't install the `ensurepip._bundled` package.

(Contributed by Victor Stinner in [bpo-42856](#).)

- Add a new `configure --without-static-libpython` option to not build the `libpythonMAJOR.MINOR.a` static library and not install the `python.o` object file.

(Contributed by Victor Stinner in [bpo-43103](#).)

- The `configure` script now uses the `pkg-config` utility, if available, to detect the location of Tcl/Tk headers and libraries. As before, those locations can be explicitly specified with the `--with-tcltk-includes` and `--with-tcltk-libs` configuration options. (Contributed by Manolis Stamatogiannakis in [bpo-42603](#).)
- Add `--with-openssl-rpath` option to `configure` script. The option simplifies building Python with a custom OpenSSL installation, e.g. `./configure --with-openssl=/path/to/openssl --with-openssl-rpath=auto`. (Contributed by Christian Heimes in [bpo-43466](#).)

13 C API Changes

13.1 PEP 652: Maintaining the Stable ABI

The Stable ABI (Application Binary Interface) for extension modules or embedding Python is now explicitly defined. stable describes C API and ABI stability guarantees along with best practices for using the Stable ABI.

(Contributed by Petr Viktorin in [PEP 652](#) and [bpo-43795](#).)

13.2 New Features

- The result of `PyNumber_Index()` now always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`. (Contributed by Serhiy Storchaka in [bpo-40792](#).)
- Add a new `orig_argv` member to the `PyConfig` structure: the list of the original command line arguments passed to the Python executable. (Contributed by Victor Stinner in [bpo-23427](#).)
- The `PyDateTime_DATE_GET_TZINFO()` and `PyDateTime_TIME_GET_TZINFO()` macros have been added for accessing the `tzinfo` attributes of `datetime.datetime` and `datetime.time` objects. (Contributed by Zackery Spytz in [bpo-30155](#).)
- Add a `PyCodec_Unregister()` function to unregister a codec search function. (Contributed by Hai Shi in [bpo-41842](#).)
- The `PyIter_Send()` function was added to allow sending value into iterator without raising `StopIteration` exception. (Contributed by Vladimir Matveev in [bpo-41756](#).)
- Add `PyUnicode_AsUTF8AndSize()` to the limited C API. (Contributed by Alex Gaynor in [bpo-41784](#).)
- Add `PyModule_AddObjectRef()` function: similar to `PyModule_AddObject()` but don't steal a reference to the value on success. (Contributed by Victor Stinner in [bpo-1635741](#).)
- Add `Py_NewRef()` and `Py_XNewRef()` functions to increment the reference count of an object and return the object. (Contributed by Victor Stinner in [bpo-42262](#).)
- The `PyType_FromSpecWithBases()` and `PyType_FromModuleAndSpec()` functions now accept a single class as the `bases` argument. (Contributed by Serhiy Storchaka in [bpo-42423](#).)

- The `PyType_FromModuleAndSpec()` function now accepts NULL `tp_doc` slot. (Contributed by Hai Shi in [bpo-41832](#).)
- The `PyType_GetSlot()` function can accept static types. (Contributed by Hai Shi and Petr Viktorin in [bpo-41073](#).)
- Add a new `PySet_CheckExact()` function to the C-API to check if an object is an instance of `set` but not an instance of a subtype. (Contributed by Pablo Galindo in [bpo-43277](#).)
- Add `PyErr_SetInterruptEx()` which allows passing a signal number to simulate. (Contributed by Antoine Pitrou in [bpo-43356](#).)
- The limited C API is now supported if Python is built in debug mode (if the `Py_DEBUG` macro is defined). In the limited C API, the `Py_IncRef()` and `Py_DecRef()` functions are now implemented as opaque function calls, rather than accessing directly the `PyObject.ob_refcnt` member, if Python is built in debug mode and the `Py_LIMITED_API` macro targets Python 3.10 or newer. It became possible to support the limited C API in debug mode because the `PyObject` structure is the same in release and debug mode since Python 3.8 (see [bpo-36465](#)).

The limited C API is still not supported in the `--with-trace-refs` special build (`Py_TRACE_REFS` macro). (Contributed by Victor Stinner in [bpo-43688](#).)

- Add the `Py_Is(x, y)` function to test if the `x` object is the `y` object, the same as `x is y` in Python. Add also the `Py_IsNone()`, `Py_IsTrue()`, `Py_IsFalse()` functions to test if an object is, respectively, the `None` singleton, the `True` singleton or the `False` singleton. (Contributed by Victor Stinner in [bpo-43753](#).)
- Add new functions to control the garbage collector from C code: `PyGC_Enable()`, `PyGC_Disable()`, `PyGC_IsEnabled()`. These functions allow to activate, deactivate and query the state of the garbage collector from C code without having to import the `gc` module.
- Add a new `Py_TPFLAGS_DISALLOW_INSTANTIATION` type flag to disallow creating type instances. (Contributed by Victor Stinner in [bpo-43916](#).)
- Add a new `Py_TPFLAGS_IMMUTABLETYPE` type flag for creating immutable type objects: type attributes cannot be set nor deleted. (Contributed by Victor Stinner and Erlend E. Aasland in [bpo-43908](#).)

13.3 Porting to Python 3.10

- The `PY_SSIZE_T_CLEAN` macro must now be defined to use `PyArg_ParseTuple()` and `Py_BuildValue()` formats which use #: es#, et#, s#, u#, y#, z#, U# and Z#. See Parsing arguments and building values and the [PEP 353](#). (Contributed by Victor Stinner in [bpo-40943](#).)
- Since `Py_REFCNT()` is changed to the inline static function, `Py_REFCNT(obj) = new_refcnt` must be replaced with `Py_SET_REFCNT(obj, new_refcnt)`: see `Py_SET_REFCNT()` (available since Python 3.9). For backward compatibility, this macro can be used:

```
#if PY_VERSION_HEX < 0x030900A4
#define Py_SET_REFCNT(obj, refcnt) ((Py_REFCNT(obj) = (refcnt)), (void)0)
#endif
```

(Contributed by Victor Stinner in [bpo-39573](#).)

- Calling `PyDict_GetItem()` without GIL held had been allowed for historical reason. It is no longer allowed. (Contributed by Victor Stinner in [bpo-40839](#).)
- `PyUnicode_FromUnicode(NULL, size)` and `PyUnicode_FromStringAndSize(NULL, size)` raise `DeprecationWarning` now. Use `PyUnicode_New()` to allocate Unicode object without initial data. (Contributed by Inada Naoki in [bpo-36346](#).)
- The private `_PyUnicode_Name_CAPI` structure of the `PyCapsule` API `unicodedata.ucnhash_CAPI` has been moved to the internal C API. (Contributed by Victor Stinner in [bpo-42157](#).)
- `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()` and `Py_GetProgramName()` functions now return NULL if called before

`Py_Initialize()` (before Python is initialized). Use the new Python Initialization Configuration API to get the Python Path Configuration.. (Contributed by Victor Stinner in [bpo-42260](#).)

- `PyList_SET_ITEM()`, `PyTuple_SET_ITEM()` and `PyCell_SET()` macros can no longer be used as l-value or r-value. For example, `x = PyList_SET_ITEM(a, b, c)` and `PyList_SET_ITEM(a, b, c) = x` now fail with a compiler error. It prevents bugs like `if (PyList_SET_ITEM (a, b, c) < 0) ... test.` (Contributed by Zackery Spytz and Victor Stinner in [bpo-30459](#).)
- The non-limited API files `odictobject.h`, `parser_interface.h`, `picklebufobject.h`, `pyarena.h`, `pyctype.h`, `pydebug.h`, `pyfpe.h`, and `pytime.h` have been moved to the `Include/cpython` directory. These files must not be included directly, as they are already included in `Python.h`: `Include Files`. If they have been included directly, consider including `Python.h` instead. (Contributed by Nicholas Sim in [bpo-35134](#))
- Use the `Py_TPFLAGS_IMMUTABLETYPE` type flag to create immutable type objects. Do not rely on `Py_TPFLAGS_HEAPTYPE` to decide if a type object is mutable or not; check if `Py_TPFLAGS_IMMUTABLETYPE` is set instead. (Contributed by Victor Stinner and Erlend E. Aasland in [bpo-43908](#).)
- The undocumented function `Py_FrozenMain` has been removed from the limited API. The function is mainly useful for custom builds of Python. (Contributed by Petr Viktorin in [bpo-26241](#))

13.4 Deprecated

- The `PyUnicode_InternImmortal()` function is now deprecated and will be removed in Python 3.12: use `PyUnicode_InternInPlace()` instead. (Contributed by Victor Stinner in [bpo-41692](#).)

13.5 Removed

- Removed `Py_UNICODE_str*` functions manipulating `Py_UNICODE*` strings. (Contributed by Inada Naoki in [bpo-41123](#).)
 - `Py_UNICODE_strlen`: use `PyUnicode_GetLength()` or `PyUnicode_GET_LENGTH`
 - `Py_UNICODE_strcat`: use `PyUnicode_CopyCharacters()` or `PyUnicode_FromFormat()`
 - `Py_UNICODE_strcpy`, `Py_UNICODE_strncpy`: use `PyUnicode_CopyCharacters()` or `PyUnicode_Substring()`
 - `Py_UNICODE_strcmp`: use `PyUnicode_Compare()`
 - `Py_UNICODE_strncmp`: use `PyUnicode_Tailmatch()`
 - `Py_UNICODE_strchr`, `Py_UNICODE strrchr`: use `PyUnicode_FindChar()`
- Removed `PyUnicode_GetMax()`. Please migrate to new ([PEP 393](#)) APIs. (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `PyLong_FromUnicode()`. Please migrate to `PyLong_FromUnicodeObject()`. (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `PyUnicode_AsUnicodeCopy()`. Please use `PyUnicode_AsUCS4Copy()` or `PyUnicode_AsWideCharString()` (Contributed by Inada Naoki in [bpo-41103](#).)
- Removed `_Py_CheckRecursionLimit` variable: it has been replaced by `ceval.recursion_limit` of the `PyInterpreterState` structure. (Contributed by Victor Stinner in [bpo-41834](#).)
- Removed undocumented macros `Py_ALLOW_RECursion` and `Py_END_ALLOW_RECursion` and the `recursion_critical` field of the `PyInterpreterState` structure. (Contributed by Serhiy Storchaka in [bpo-41936](#).)

- Removed the undocumented `PyOS_InitInterrupts()` function. Initializing Python already implicitly installs signal handlers: see `PyConfig.install_signal_handlers`. (Contributed by Victor Stinner in [bpo-41713](#).)
- Remove the `PyAST_Validate()` function. It is no longer possible to build a AST object (`mod_ty` type) with the public C API. The function was already excluded from the limited C API ([PEP 384](#)). (Contributed by Victor Stinner in [bpo-43244](#).)
- Remove the `symtable.h` header file and the undocumented functions:
 - `PyST_GetScope()`
 - `PySymtable_Build()`
 - `PySymtable_BuildObject()`
 - `PySymtable_Free()`
 - `Py_SymtableString()`
 - `Py_SymtableStringObject()`

The `Py_SymtableString()` function was part the stable ABI by mistake but it could not be used, because the `symtable.h` header file was excluded from the limited C API.

Use Python `symtable` module instead. (Contributed by Victor Stinner in [bpo-43244](#).)

- Remove `PyOS_ReadlineFunctionPointer()` from the limited C API headers and from `python3.dll`, the library that provides the stable ABI on Windows. Since the function takes a `FILE*` argument, its ABI stability cannot be guaranteed. (Contributed by Petr Viktorin in [bpo-43868](#).)
- Remove `ast.h`, `asdl.h`, and `Python-ast.h` header files. These functions were undocumented and excluded from the limited C API. Most names defined by these header files were not prefixed by `Py` and so could create names conflicts. For example, `Python-ast.h` defined a `Yield` macro which was conflict with the `Yield` name used by the Windows `<winbase.h>` header. Use the Python `ast` module instead. (Contributed by Victor Stinner in [bpo-43244](#).)
- Remove the compiler and parser functions using `struct _mod` type, because the public AST C API was removed:
 - `PyAST_Compiler()`
 - `PyAST_CompilerEx()`
 - `PyAST_CompilerObject()`
 - `PyFuture_FromAST()`
 - `PyFuture_FromASTObject()`
 - `PyParser_ASTFromFile()`
 - `PyParser_ASTFromFileObject()`
 - `PyParser_ASTFromFilename()`
 - `PyParser_ASTFromString()`
 - `PyParser_ASTFromStringObject()`

These functions were undocumented and excluded from the limited C API. (Contributed by Victor Stinner in [bpo-43244](#).)

- Remove the `pyarena.h` header file with functions:

- `PyArena_New()`
- `PyArena_Free()`
- `PyArena_Malloc()`
- `PyArena_AddPyObject()`

These functions were undocumented, excluded from the limited C API, and were only used internally by the compiler. (Contributed by Victor Stinner in [bpo-43244](#).)

- The `PyThreadState.use_tracing` member has been removed to optimize Python. (Contributed by Mark Shannon in [bpo-43760](#).)

Index

E

environment variable
 PYTHONTREADDEBUG, [27](#)
 PYTHONWARNDEFAULTENCODING, [12](#)

P

Python Enhancement Proposals

 PEP 353, [31](#)
 PEP 384, [33](#)
 PEP 393, [32](#)
 PEP 451, [24](#)
 PEP 484, [12](#), [13](#)
 PEP 526, [14](#)
 PEP 586, [22](#)
 PEP 590, [24](#)
 PEP 597, [3](#)
 PEP 604, [3](#), [13](#)
 PEP 612, [3](#), [13](#)
 PEP 613, [3](#), [13](#)
 PEP 617, [4](#)
 PEP 618, [3](#), [14](#)
 PEP 623, [3](#)
 PEP 624, [3](#)
 PEP 626, [3](#)
 PEP 632, [3](#), [17](#)
 PEP 634, [3](#), [12](#)
 PEP 635, [3](#), [12](#)
 PEP 636, [3](#), [12](#)
 PEP 644, [3](#), [18](#), [21](#), [29](#)
 PEP 647, [13](#)
 PEP 652, [30](#)

PYTHONTREADDEBUG, [27](#)
PYTHONWARNDEFAULTENCODING, [12](#)

R

RFC
 RFC 3986, [23](#)